

# Haskell, eine rein funktionale Programmiersprache

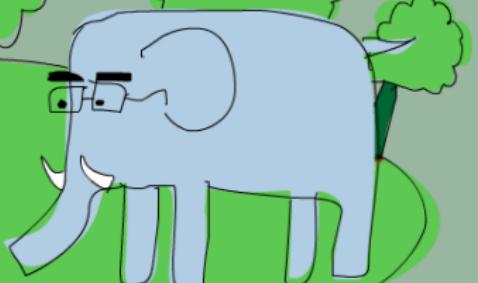
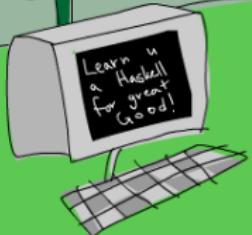
Ingo Blechschmidt  
<iblech@web.de>

Augsburg.pm  
3. Juni 2014





Learn You a Haskell  
for Great  
Good!



Fragen zu jeder Zeit willkommen!

## 1 Grundlegendes

- Keine Variablen
- Keine Nebeneffekte
- Keine Ein-/Ausgabe
- Erste Beispiele: Aufsummieren und Quicksort

## 2 Typsystem

- Grundtypen
- Typerschließung
- Aktionstypen

## 3 Weitere Fähigkeiten

- Aktionsarten
- Parser-Aktionen
- Bedarfsauswertung
- QuickCheck
- Webprogrammierung

## 4 Gemeinschaft

- Hackage
- Größere Anwendungen
- Einstiegspunkte

# Über Haskell

- Hochsprache, rein funktional, statisch stark typisiert, „nicht-strikt“
- Sprachspezifikation durch ein Komitee
- Veröffentlichung von Haskell 1.0: 1990
- Kompiliert und interpretiert
- REPL



# Imperative Sprachen

Kennzeichen imperativer Sprachen:

- Anweisungen
- veränderliche Variablen
- Nebeneffekte

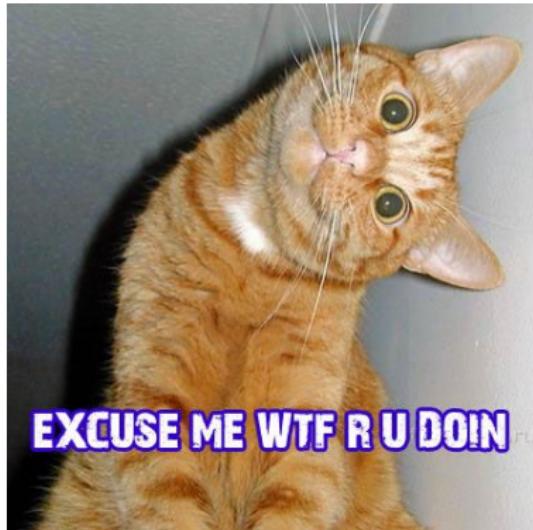


# Haskell ist komisch!

Haskell ist rein funktional:

- keine Anweisungen
- keine veränderliche Variablen
- keine Nebeneffekte





**EXCUSE ME WTF R U DOIN**



**ONE DOES NOT SIMPLY**

**"DO STUFF" IN HASKELL**

# Erstes Beispiel

```
# Perl
sub main {
    my $radius      = 42;
    my $quadriert = $radius ** 2;
    my $flaeche    = $quadriert * pi;
    print $flaeche;
}
```



# Erstes Beispiel

```
# Perl
sub main {
    my $radius      = 42;
    my $quadriert = $radius ** 2;
    my $flaeche    = $quadriert * pi;
    print $flaeche;
}
```

-- Haskell

```
main =
    let radius      = 42
        quadriert = radius^2
        flaeche    = quadriert * pi
    in print flaeche
```



# Erstes Beispiel

```
# Perl
sub main {
    my $radius      = 42;
    my $quadriert = $radius ** 2;
    my $flaeche    = $quadriert * pi;
    print $flaeche;
}

-- Haskell
main =
    let flaeche      = quadriert * pi
        quadriert = radius^2
        radius     = 42
    in print flaeche
```



# Erstes Beispiel

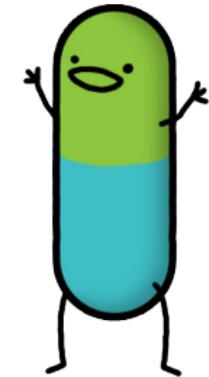
```
# Perl
sub main {
    my $radius      = 42;
    my $quadriert = $radius ** 2;
    my $flaeche    = $quadriert * pi;
    print $flaeche;
}
```

```
-- Haskell
main = print flaeche
      where
          flaeche    = quadriert * pi
          quadriert = radius^2
          radius    = 42
```



# Keine Nebeneffekte

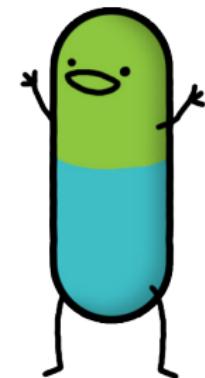
```
# Perl
my $a = f($x);
my $b = f($x);
...
# Ist $a == $b?
```



# Keine Nebeneffekte

```
# Perl
my $a = f($x);
my $b = f($x);
...
# Ist $a == $b?
```

```
-- Haskell
let a = f x
    b = f x
in ...
-- a == b gilt stets.
```

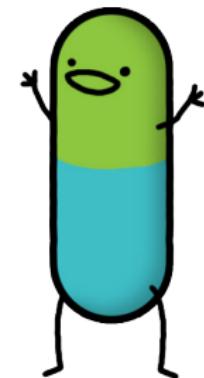


# Keine Nebeneffekte

```
# Perl
my $a = f($x);
my $b = f($x);
...
# Ist $a == $b?
```

```
-- Haskell
let a = f x
    b = f x
in ...
-- a == b gilt stets.
```

- Gleiche Argumente  $\rightsquigarrow$  gleiche Rückgaben
- Keine Ein-/Ausgabe,  
keine Zustandsveränderungen, ...
- Rein lokales Codeverständnis!
- Tiefgreifende Optimierungsmöglichkeiten!



# Keine Eingabe/Ausgabe

```
let x = getLine  
in print (x ++ x)
```

vs.

```
print (getLine ++ getLine)
```

# Keine Eingabe/Ausgabe

```
-- Compile-Zeit-Fehler!
let x = getLine
in print (x ++ x)
```

vs.

```
-- Compile-Zeit-Fehler!
print (getLine ++ getLine)
```



# Keine Eingabe/Ausgabe

```
-- Compile-Zeit-Fehler!
let x = getLine
in print (x ++ x)
```

vs.

```
-- Compile-Zeit-Fehler!
print (getLine ++ getLine)
```

- Aktionen  $\neq$  Werte
- Explizite Unterscheidung durchs Typsystem (s. gleich)



# Beispiel: Aufsummieren einer Zahlenliste

Deklaration durch Musterabgleich:

```
sum []      = 0
sum (x:xs) = x + sum xs
```

Beispielauswertung:

```
sum [1,2,3]
== 1 + sum [2,3]
== 1 + (2 + sum [3])
== 1 + (2 + (3 + sum []))
== 1 + (2 + (3 + 0))
== 6
```



# Beispiel: Quicksort

```
quicksort []      = []
quicksort (x:xs) =
    quicksort kleinere ++ [x] ++ quicksort groessere
    where
        kleinere  = [ y  |  y <- xs,  y <  x ]
        groessere = [ y  |  y <- xs,  y >= x ]
```



# Werte und Typen

- Statische starke Typisierung;  
jeder Wert ist von genau einem Typ.
- Keine impliziten Typumwandlungen
- Enorm hilfreich!



# Werte und Typen

- Statische starke Typisierung;  
jeder Wert ist von genau einem Typ.
- Keine impliziten Typumwandlungen
- Enorm hilfreich!
- Primitive Typen:

`"Hallo, Welt!" :: String`

`True :: Bool`

`37 :: Integer` (beliebige Größe)

`37 :: Int` (mind. 31 Bit)

- Zusammengesetzte Typen:

`['A', 'B', 'C'] :: [Char]`

`[[1,2], [3], []] :: [[Integer]]`



# Werte und Typen (Forts.)

## ■ Funktionstypen:

```
head      :: [a]  -> a
-- Bsp.: head [1,2,3] == 1
```



```
tail      :: [a]  -> [a]
-- Bsp.: tail [1,2,3] == [2,3]
```

-- Operatoren:

```
(&&)      :: Bool -> Bool -> Bool
(++)      :: [a]  -> [a]  -> [a]
```

# java.io.InputStreamReader



# Automatische Typerschließung

Automatische Erschließung nicht angegebener Typen durch den Compiler

```
greet name = "Hallo " ++ name ++ "!"
```



# Automatische Typerschließung

Automatische Erschließung nicht angegebener Typen durch den Compiler

```
greet :: String -> String
greet name = "Hallo " ++ name ++ "!"
```



# Automatische Typerschließung

Automatische Erschließung nicht angegebener Typen durch den Compiler

```
greet :: String -> String
greet name = "Hallo " ++ name ++ "!"
```

```
dup xs = xs ++ xs
-- Bsp.:
dup [1,2,3] ==
[1,2,3,1,2,3]
```



# Automatische Typerschließung

Automatische Erschließung nicht angegebener Typen durch den Compiler

```
greet :: String -> String
greet name = "Hallo " ++ name ++ "!"
```

```
dup :: [a] -> [a]
dup xs = xs ++ xs
-- Bsp.:
dup [1,2,3] ==
[1,2,3,1,2,3]
```



# Typen von Ein-/Ausgabe-Operationen

- `IO Foo` meint:  
Wert vom Typ `Foo` produzierende IO-Aktion
- Häufig benutzte Ein-/Ausgabe-Operationen:

```
getLine  :: IO String
putStr   :: String    -> IO ()
readFile :: FilePath  -> IO String
```

# Typen von Ein-/Ausgabe-Operationen

- `IO Foo` meint:  
Wert vom Typ `Foo` produzierende IO-Aktion
- Häufig benutzte Ein-/Ausgabe-Operationen:

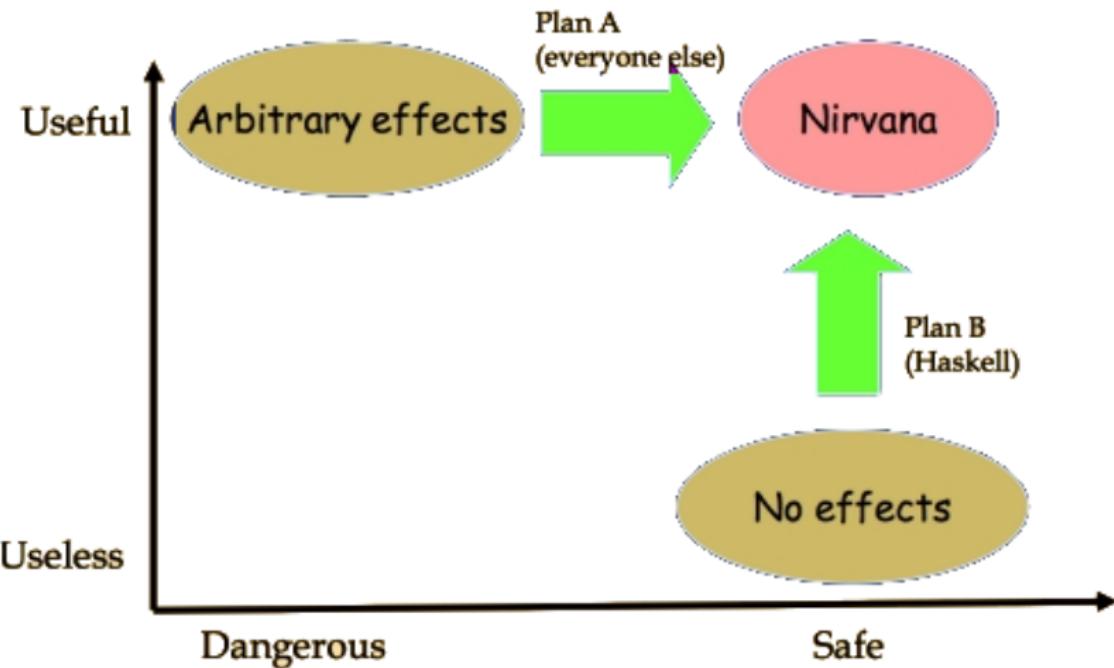
```
getLine  :: IO String
putStr   :: String    -> IO ()
readFile :: FilePath  -> IO String
```

- Zum früheren Beispiel:

```
main :: IO ()
main = do
    x <- getLine
    print (x ++ x)
```

```
main :: IO ()
main = do
    x <- getLine
    y <- getLine
    print (x ++ y)
```

# The challenge of effects



Grafik gestohlen von: Simon Peyton Jones

# Arten von Aktionen

- IO (Eingabe/Ausgabe)
- Parser
- Reader (vererbende Umgebung)
- State (veränderlicher Zustand)
- Writer (Logging)
- Listen für Nichtdeterminismus und Logikprogrammierung
- Cont (Continuations)
- ...



# Parser-Aktionen

Parser für Perl-Bezeichner (z.B. \$foo, @bar123):

```
perlBezeichner :: Parser String
perlBezeichner = do
    sigil <- oneOf "$@%&"
    name  <- many alphaNum
    return (sigil : name)
```

Dabei verwendete Funktionen aus  
Parser-Bibliothek:

```
oneOf      :: [Char] -> Parser Char
alphaNum   :: Parser Char
many       :: Parser a -> Parser [a]
```



# Bedarfsauswertung

```
-- 100. Zeile aus Datei ausgeben
main = do
    daten <- readFile "große-datei.txt"
    print (lines daten !! 99)
```



# Bedarfsauswertung

```
-- 100. Zeile aus Datei ausgeben
main = do
    daten <- readFile "große-datei.txt"
    print (lines daten !! 99)
```

- Auswertung von Ausdrücken erst dann, wenn Ergebnisse wirklich benötigt
- Wegabstraktion des Speicherkonzepts!
- Somit Fähigkeit für unendliche Datenstrukturen: Potenzreihen, Zeitreihen, Entscheidungsbäume, ...



# Bedarfsauswertung (Forts.)

```
natürlicheZahlen = [1..]  
-- [1, 2, 3, 4, ...]
```



# Bedarfsauswertung (Forts.)

```
natürlicheZahlen = [1..]  
-- [1, 2, 3, 4, ...]
```

```
ungeradeZahlen = filter odd [1..]  
-- [1, 3, 5, 7, ...]
```



# Bedarfsauswertung (Forts.)

```
natürlicheZahlen = [1..]  
-- [1, 2, 3, 4, ...]
```

```
ungeradeZahlen = filter odd [1..]  
-- [1, 3, 5, 7, ...]
```

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)  
-- [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...]
```



# QuickCheck

```
# Perl
is(sqrt(0), 0, "sqrt(0) ist ok");
is(sqrt(1), 1, "sqrt(1) ist ok");
is(sqrt(4), 2, "sqrt(4) ist ok");
is(sqrt(9), 3, "sqrt(9) ist ok");
is(sqrt(16), 4, "sqrt(16) ist ok");
...; # ??
```



# QuickCheck

```
# Perl
is(sqrt(0), 0, "sqrt(0) ist ok");
is(sqrt(1), 1, "sqrt(1) ist ok");
is(sqrt(4), 2, "sqrt(4) ist ok");
is(sqrt(9), 3, "sqrt(9) ist ok");
is(sqrt(16), 4, "sqrt(16) ist ok");
...; # ??
```

```
-- Haskell mit QuickCheck
propSqrt :: Double -> Bool
propSqrt x =
    sqrt (x * x) == x
```



# QuickCheck (Forts.)

```
propSqrt :: Double -> Bool
propSqrt x = sqrt (x * x) == x
```

```
ghci> quickCheck propSqrt
Falsifiable, after 6 tests:
-4
```



# QuickCheck (Forts.)

```
propSqrt :: Double -> Bool
propSqrt x = sqrt (x * x) == x
```

```
ghci> quickCheck propSqrt
Falsifiable, after 6 tests:
-4
```

```
propSqrt' :: Double -> Bool
propSqrt' x = sqrt (x * x) == abs x
```

```
ghci> quickCheck propSqrt'
OK, passed 100 tests.
```



# QuickCheck (Forts.)

- Spezifikationsüberprüfung durch zufällig generierte Stichproben
- Enorm hilfreich!



# QuickCheck (Forts.)

- Spezifikationsüberprüfung durch zufällig generierte Stichproben
  - Enorm hilfreich!
- 
- Implementierung durch Typklasse:

```
class Arbitrary a where
    arbitrary :: Gen a
```
  - Instanzen von `Arbitrary` nicht nur für primitive, sondern automatisch auch für zusammengesetzte Typen



# Webprogrammierung

- Wieso Haskell fürs Web?  
Effizienz, Sicherheit & die üblichen Vorteile
- Webframeworks: Happstack, Snap, Yesod
- Leichtgewichtige Threads, asynchrone IO
- Minimalbeispiel:

```
mainLoop sock = forever $ do
    conn <- accept sock
    forkIO $ runConn conn
```

[http://www.haskell.org/haskellwiki/Implement\\_a\\_chat\\_server](http://www.haskell.org/haskellwiki/Implement_a_chat_server)

# Paketarchiv Hackage

- Entstehung 2007, mittlerweile 5000+ Pakete
  - Installationswerkzeug cabal-install

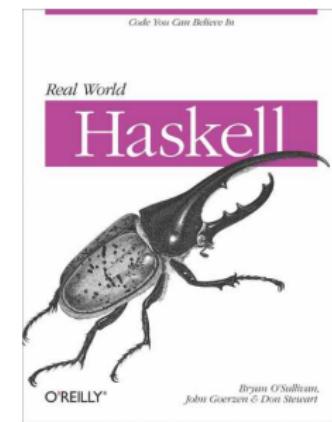
ai algorithms bioinformatics codec compilers  
concurrency console control cryptography data  
database datastructures debug  
development distributedcomputing distribution  
failure ffi foreign frp game generics graphics gui  
hardware interpreters language math monads  
music network nlp numerical parsing sound  
system testing text theoremprovers userinterfaces  
utils web xml

# Größere Anwendungen

- Glasgow Haskell Compiler (GHC)
- darcs, verteiltes Versionskontrollsystem
- xmonad, „tiling“ Fenstermanager
- Pugs, Perl-6-Prototyp
- Cryptol, Sprache für Kryptographie
- Criterion, Benchmarktoolkit



- <http://haskell.org/>
- zum Herumspielen im Browser:  
<http://tryhaskell.org/>
- interaktive Haskell-Shell:  
\$ apt-get install ghc  
\$ ghci
- <http://learnyouahaskell.com/>
- Buch: Real World Haskell, O'Reilly
- Teile dieses Vortrags inspiriert von einem Vortrag von Audrey Tang:  
<http://feather.perl6.nl/~audreyt/osdc/haskell.xul>





Thank You,  
Thank You,

► Bonusfolien



# Bonusfolien

## 5 Details zum Typsystem

- Typklassen
- Benutzerdefinierte Datentypen
- Umgang mit fehlenden Werten

## 6 Foreign Function Interface

## 7 Nebenläufigkeit

- Software Transactional Memory
- Data Parallel Haskell

## 8 Bildquellen



# Typklassen (= Schnittstellen, Rollen)

## ■ Typklassen für ad-hoc Polymorphismus:

```
add37 x = x + 37
-- Bsp.: add37 5 == 42
```

# Typklassen (= Schnittstellen, Rollen)

## ■ Typklassen für ad-hoc Polymorphismus:

```
add37 :: a -> a
```

```
add37 x = x + 37
```

```
-- Bsp.: add37 5 == 42
```

# Typklassen (= Schnittstellen, Rollen)

## ■ Typklassen für ad-hoc Polymorphismus:

```
add37 :: (Num a) => a -> a
```

```
add37 x = x + 37
```

```
-- Bsp.: add37 5 == 42
```

# Typklassen (= Schnittstellen, Rollen)

## ■ Typklassen für ad-hoc Polymorphismus:

```
add37 :: (Num a) => a -> a
```

```
add37 x = x + 37
```

```
-- Bsp.: add37 5 == 42
```

```
min x y = if x <= y then x else y
```

```
-- Bsp.: min 19 17 == 17
```

# Typklassen (= Schnittstellen, Rollen)

## ■ Typklassen für ad-hoc Polymorphismus:

```
add37 :: (Num a) => a -> a
```

```
add37 x = x + 37
```

```
-- Bsp.: add37 5 == 42
```

```
min :: (Ord a) => a -> a -> a
```

```
min x y = if x <= y then x else y
```

```
-- Bsp.: min 19 17 == 17
```

# Typklassen (= Schnittstellen, Rollen)

## ■ Typklassen für ad-hoc Polymorphismus:

```
add37 :: (Num a) => a -> a
```

```
add37 x = x + 37
```

```
-- Bsp.: add37 5 == 42
```

```
min :: (Ord a) => a -> a -> a
```

```
min x y = if x <= y then x else y
```

```
-- Bsp.: min 19 17 == 17
```

## ■ Deklaration:

```
class Num a where
```

```
    (+) :: a -> a -> a
```

```
    (-) :: a -> a -> a
```

```
    (*) :: a -> a -> a
```

```
    ...
```

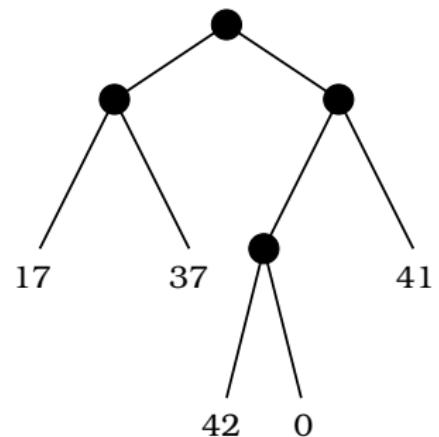
# Benutzerdefinierte Datentypen

```
data Tree = Leaf Int | Fork Tree Tree
```

Konstruktoren:

Leaf :: Int  $\rightarrow$  Tree und

Fork :: Tree  $\rightarrow$  Tree  $\rightarrow$  Tree



# Benutzerdefinierte Datentypen

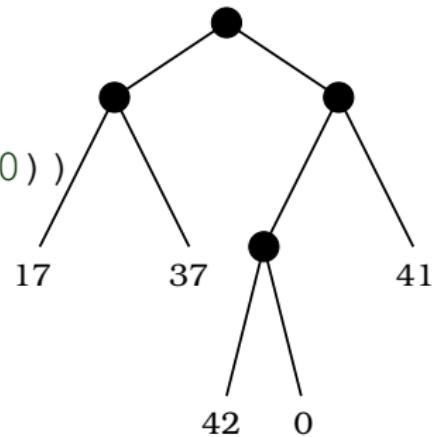
```
data Tree = Leaf Int | Fork Tree Tree
```

Konstruktoren:

Leaf :: Int  $\rightarrow$  Tree und

Fork :: Tree  $\rightarrow$  Tree  $\rightarrow$  Tree

```
beispielBaum = Fork
  (Fork (Leaf 17) (Leaf 37))
  (Fork
    (Fork (Leaf 42) (Leaf 0))
    (Leaf 41))
```



# Benutzerdefinierte Datentypen

```
data Tree = Leaf Int | Fork Tree Tree
```

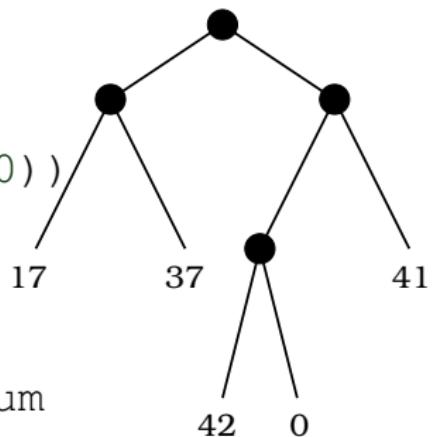
Konstruktoren:

Leaf :: Int  $\rightarrow$  Tree und

Fork :: Tree  $\rightarrow$  Tree  $\rightarrow$  Tree

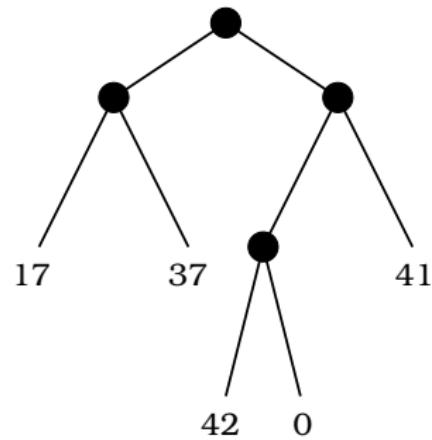
```
beispielBaum = Fork
  (Fork (Leaf 17) (Leaf 37))
  (Fork
    (Fork (Leaf 42) (Leaf 0))
    (Leaf 41))
```

```
komischerBaum =
  Fork (Leaf 23) komischerBaum
```



# Benutzerdefinierte Datentypen (Forts.)

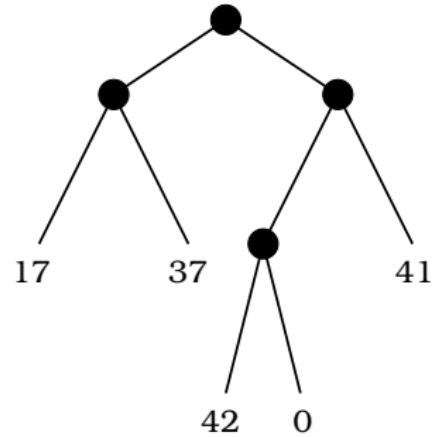
```
data Tree    = Leaf Int | Fork Tree Tree
```



# Benutzerdefinierte Datentypen (Forts.)

```
data Tree = Leaf Int | Fork Tree Tree
```

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
```



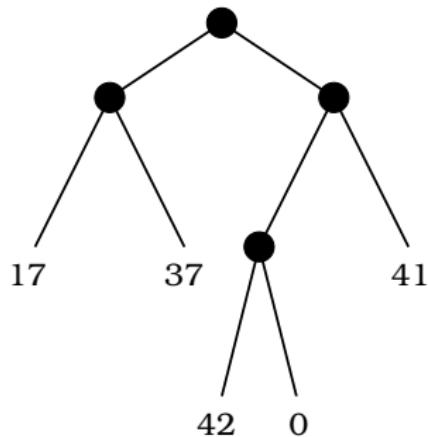
# Benutzerdefinierte Datentypen (Forts.)

```
data Tree = Leaf Int | Fork Tree Tree
```

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
```

-- Gesamtzahl Blätter zählen

```
size :: Tree a -> Integer
size (Leaf _) = 1
size (Fork links rechts) =
    size links + size rechts
```



# Benutzerdefinierte Datentypen (Forts.)

```
data Tree = Leaf Int | Fork Tree Tree
```

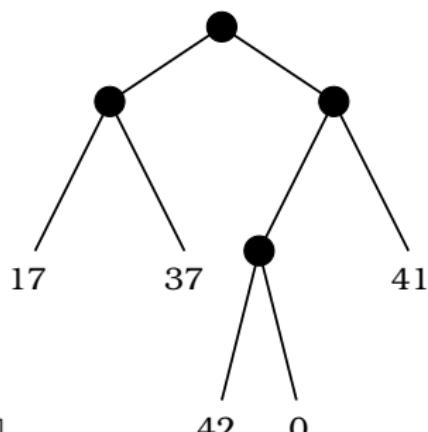
```
data Tree a = Leaf a | Fork (Tree a) (Tree a)
```

-- Gesamtzahl Blätter zählen

```
size :: Tree a -> Integer
size (Leaf _) = 1
size (Fork links rechts) =
    size links + size rechts
```

-- Blätter als Liste zurückgeben

```
inorder :: Tree a -> [a]
inorder (Leaf x) = [x]
inorder (Fork links rechts) =
    inorder links ++ inorder rechts
```



# Umgang mit fehlenden Werten

- „Null References: The Billion Dollar Mistake“ (Tony Hoare)
- In Haskell: Explizite Kennzeichnung von möglicherweise fehlenden Werten durch Maybe-Typen



```
data Maybe a = Nothing | Just a
```

```
lookupFarbe :: String -> Maybe Color
-- Bsp.:
lookupFarbe "Simpsons" == Just YELLOW
lookupFarbe "Simqsons" == Nothing
```

# Maybe-Ketten

Anwendbarkeit des syntaktischen Zuckers von Aktionstypen für Maybe:

```
berechneA :: Integer -> Maybe String
berechneB :: Integer -> Maybe [Double]
berechneC :: [Double] -> Maybe String
```

```
berechne :: Integer -> Maybe String
berechne x = do
    teil1  <- berechneA x
    teil2  <- berechneB x
    teil2' <- berechneC teil2
    return (teil1 ++ teil2)
```

# Foreign Function Interface

```
{-# INCLUDE <math.h> #-}  
foreign import ccall unsafe "sin"  
    c_sin :: CDouble -> CDouble
```



- Einbindung von (C-)Bibliotheken
- Keine besondere Handhabung der importierten Funktionen
- Callbacks aus dem C-Code heraus möglich

# Nebenläufigkeit

- Aufgabenparallelismus:  
Software Transactional Memory [► weiter](#)
- Datenparallelismus:  
Data Parallel Haskell [► weiter](#)



# Traditionelles Locking

- Schwierigkeiten bei traditionellem Locking:  
nichtlokales Denken, Deadlocks, Livelocks,  
Prioritätsinversion
- „lock-based programs do not compose“



# Traditionelles Locking (Forts.)

```
1 # Perl
$x->withdraw(3);
$y->deposit(3);
# Race Condition!
```



# Traditionelles Locking (Forts.)

```
1 # Perl
  $x->withdraw(3);
  $y->deposit(3);
  # Race Condition!

2 $x->lock(); $y->lock();
  $x->withdraw(3);
  $y->deposit(3);
  $y->unlock(); $x->unlock();
```



# Traditionelles Locking (Forts.)

1 # Perl  
\$x->withdraw(3);  
\$y->deposit(3);  
# Race Condition!

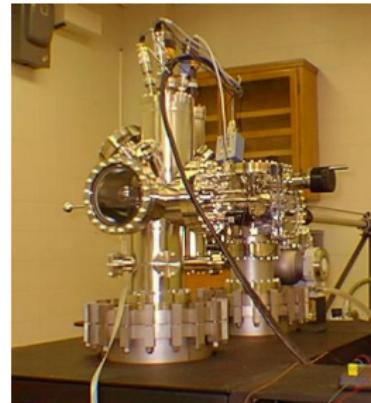
2 \$x->lock(); \$y->lock();  
\$x->withdraw(3);  
\$y->deposit(3);  
\$y->unlock(); \$x->unlock();

3 { \$x->lock(); \$y->lock(); ...; }  
vs.  
{ \$y->lock(); \$x->lock(); ...; }  
# Deadlock!



# Software Transactional Memory

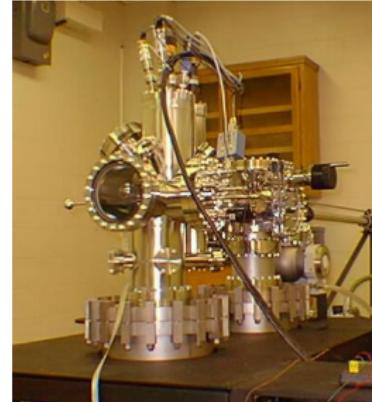
- Transaktionsvorstellung:  
Unabhängige, einfädige Ausführung jeder Transaktion
- Implementierung durch Schreiben in temporären Puffer; Commit nur, wenn gelesene Variablen noch unverändert, sonst Neustart
- Vorteile:  
keine expliziten Locks,  
keine Race Conditions,  
Komponierbarkeit



# Software Transactional Memory (Forts.)

```
-- Haskell
withdraw :: Account -> Int -> STM ()
deposit  :: Account -> Int -> STM ()

atomically $ do
    withdraw x 3
    deposit  y 3
```



# Flat/Nested Data Parallelism

- Flat Data Parallelism:

```
for my $i (1..$N) {  
    work($i);  
}
```

- Parallelisierung durch einfaches Aufteilen der Fälle auf die Prozessoren
- Effizienz verschenkt, wenn `work()` selbst parallel arbeiten könnte!

# Data Parallel Haskell

- Umsetzung von Nested Data Parallelism
- Automatische Programmtransformationen:  
Flattening (global), Fusion (lokal)
- Wesentliche Abhängigkeit von Haskells reiner Funktionalität

# Data Parallel Haskell

- Umsetzung von Nested Data Parallelism
- Automatische Programmtransformationen:  
Flattening (global), Fusion (lokal)
- Wesentliche Abhängigkeit von Haskells reiner Funktionalität

```
-- Komponentenweise Vektor-Addition
addP :: (Num a) => [:a:] -> [:a:] -> [:a:]
addP xs ys =
  [: x + y | x <- xs | y <- ys :]

-- Skalarprodukt
dotP :: (Num a) => [:a:] -> [:a:] -> a
dotP xs ys = sumP (mulP xs ys)
```

# Bildquellen

- <http://betriebsrat.files.wordpress.com/2009/10/zeitungs-logo-standard.jpg>
- [http://de.academic.ru/pictures/dewiki/100/dining\\_philosophers.png](http://de.academic.ru/pictures/dewiki/100/dining_philosophers.png)
- <http://i.ehow.com/images/a04/8i/g0/show-real-smile-photographs-800X800.jpg>
- [http://imgs.xkcd.com/comics/regular\\_expressions.png](http://imgs.xkcd.com/comics/regular_expressions.png)
- <http://learnyouahaskell.com/splash.png>
- <http://media.nokrev.com/junk/haskell-logos/logo7.png>
- <http://otierney.net/images/perl6.gif>
- <http://perl.plover.com/yak/presentation/samples/present.gif>
- <http://save-endo.cs.uu.nl/target.png>
- <http://theleftwinger.files.wordpress.com/2010/01/lazy.jpg>
- <http://upload.wikimedia.org/wikipedia/commons/7/7d/Bug.png>
- [http://upload.wikimedia.org/wikipedia/commons/c/c6/Metal\\_movable\\_type\\_edit.jpg](http://upload.wikimedia.org/wikipedia/commons/c/c6/Metal_movable_type_edit.jpg)
- <http://upload.wikimedia.org/wikipedia/commons/d/d0/Jabberwocky.jpg>
- <http://wordaligned.org/images/the-challenge-of-effects.jpg>
- <http://www.coverbrowser.com/image/oreilly-books/42-1.jpg>
- <http://www.ctri.co.uk/images/cat30.jpg>
- <http://www.darcs.net/logos/logo.png>
- [http://www.fos.org.au/custom/files/media/sort\\_it\\_cover\\_300.jpg](http://www.fos.org.au/custom/files/media/sort_it_cover_300.jpg)
- <http://www.galois.com/~dons/images/cloud.png>
- [http://www.galois.com/files/Cryptol/Cryptol\\_logo\\_image.png](http://www.galois.com/files/Cryptol/Cryptol_logo_image.png)
- <http://www.homemortgagerates.us/variable-rates-636.jpg>
- [http://www.lakehousecreations.com/images/ThankYou/Thank%20You%202003%20\(12\).jpg](http://www.lakehousecreations.com/images/ThankYou/Thank%20You%202003%20(12).jpg)
- <http://www.nataliedee.com/093009/death-is-a-side-effect-of-most-things.jpg>
- <http://www.quickmeme.com/img/bd/bdb89c032c3c7c09a86897422bf3bd32a0bb65731c7fd38ba5ea67741621ad35.jpg>
- [http://www.simpsonsonline.com/rb/fakten/vorspann/02\\_kasse.jpg](http://www.simpsonsonline.com/rb/fakten/vorspann/02_kasse.jpg)
- [http://www.sketchybeast.com/wp-content/uploads/2007/12/input\\_output.jpg](http://www.sketchybeast.com/wp-content/uploads/2007/12/input_output.jpg)
- <http://www.sopaed-lern.uni-wuerzburg.de/uploads/pics/Pruefung.jpg>
- [http://www.wfu.edu/nanotech/Microscopy%20Facility/stm\\_view.jpg](http://www.wfu.edu/nanotech/Microscopy%20Facility/stm_view.jpg)
- <http://www.wishlist.nu/wp-content/uploads/2007/10/gargoyle.jpg>
- <http://xmonad.org/images/logo.png>